INTRODUZIONE

- Gli array presentano due limitazioni:
  - Modificare la dimensione dell'array richiede la creazione di un nuovo array e la copia di tutti i dati dall'array originale a quello nuovo.
  - Inserire un elemento nell'array richiede lo scorrimento di dati nell'array.

- Queste limitazioni possono essere superate da strutture concatenate: una collezione di nodi che contengono dati e riferimenti ad altri nodi. In particolare si fa riferimento alle *liste concatenate* (*linked list*).

INTRODUZIONE

- Uno svantaggio delle liste concatenate consiste nell'onerosità dell'accesso ai suoi elementi: l'unico modo per raggiungere un dato elemento della lista è quello di seguire le connessioni della lista dall'inizio in modo *sequenziale*. Mentre negli array è possibile accedere a qualsiasi elemento attraverso un indice.
- I metodi di manipolazione delle liste sono diversi da quelli visti per gli array.
- In generale, una lista rende più semplice la riorganizzazione degli elementi.

LIST: specifica

- Una possibile interfaccia è la seguente:

```
import java.util.*;
public interface List {
  boolean add(Object x);//inserisce l'oggetto x
                          in fondo alla lista
  boolean remove(Object x);//cancella la prima
                         occorrenza dell'oggetto x
  boolean contains(Object x);//verifica se la
           lista contiene l'oggetto x
  boolean isEmpty();// verifica se lista è
                          logicamente vuota
  void clear();// svuota la lista
  Iterator iterator(); //restituisce un iteratore
             per scandire gli elementi in sequenza
}
```


LIST

Sia `L` un oggetto di tipo `List`: ecco alcune operazioni senza iteratori.

Invocazione metodo	Stato della lista	risultato
`L.isEmpty();`	[]	true
`L.add(new Character('a'));`	[a]	true
`L.add(new Character('b'));`	[a b]	true
`L.add(new Character('c'));`	[a b c]	true
`L.contains(new Character('b');`	[a b c]	true
`L.remove(new Character('b');`	[a c]	true
`L.contains(new Character('b');`	[a c]	false
`L.isEmpty();`	[a c]	false

LIST

Alcune operazioni con iteratori: la barra | denota un cursore.

Invocazione metodo	Stato della lista	risultato
`L.add(new Integer(2));`	[2]	true
`L.add(new Integer(5));`	[2 5]	true
`L.add(new Integer(4));`	[2 5 4]	true
`Iterator i = L.iterator();`	[| 2 5 4]	i è un iteratore
`i.next()`	[2 | 5 4]	2
`i.next()`	[2 5 | 4]	5
`i.remove()`	[2 | 4]	
`i.hasNext()`	[2 | 4]	true

LIST: esempio

- Una porzione di codice che legge un file (`nomi.txt`) e memorizza il contenuto in una lista. Viene allocata la giusta quantità di elementi. Stampa il contenuto della lista attraverso l'uso di un iteratore. Rimuove un elemento della lista (`Maria`) e stampa il nuovo contenuto della lista.

nomi.txt

Luca
Maria
Elisa
Andrea

output

Luca Maria Elisa Andrea
Luca Elisa Andrea

LIST: esempio

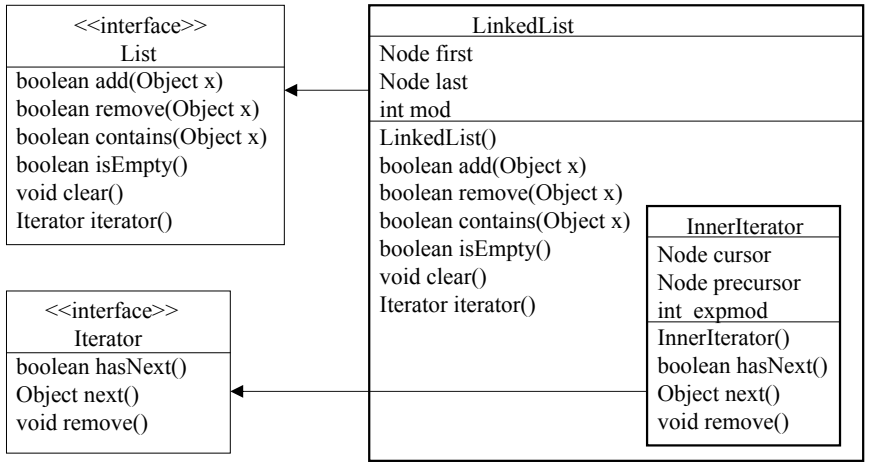
```
List L =new LinkedList();
BufferedReader in = new BufferedReader(new
                      FileReader("nomi.txt"));
String str;
while( (str=in.readLine()) != null)
          L.add(str);
in.close();

for(Iterator i=L.iterator();i.hasNext(); )
    System.out.print(i.next()+" ");
L.remove("Maria");
System.out.println();
for(Iterator i=L.iterator();i.hasNext(); )
    System.out.print(i.next()+" ");
```


LIST: implementazione

- Una possibile realizzazione della lista consiste nel considerare una sequenza di nodi che contengono un dato e un riferimento al nodo successivo nella sequenza. Tale sequenza è realizzata nella classe `LinkedList` che implementa l'interfaccia `List`.
- Un oggetto di tipo `LinkedList` è manipolato utilizzando i metodi resi disponibili dall'interfaccia `List`, non si accede direttamente alla sequenza di nodi. È un *tipo di dato astratto*, ADT.

LIST: implementazione

Si realizza l'interfaccia `List` con la classe `LinkedList` e l'interfaccia `Iterator` con la classe interna `InnerIterator`.



NODI: operazioni principali

- Una possibile definizione di nodo per l'implementazione della lista è la seguente:

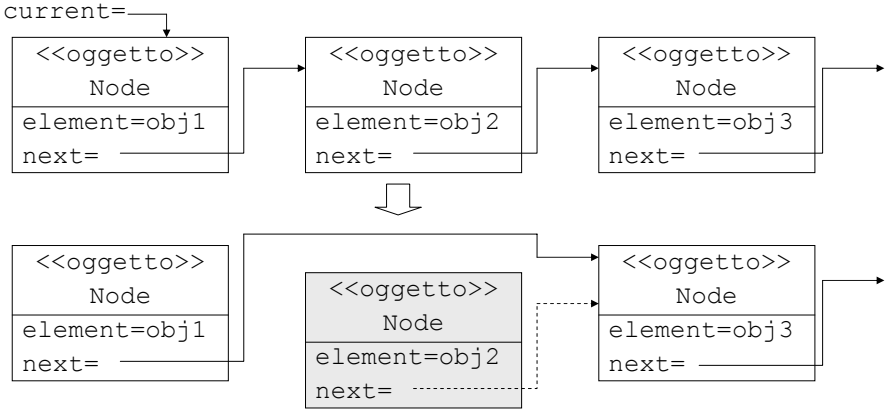
```
public class Node {
  Object element;
  Node next;
  public Node(Object element){
   this(element,null);
  }
  public Node(Object element, Node next){
   if (element == null)
         throw new IllegalArgumentException();
   this.element=element;
   this.next=next;
}}
```


NODI: operazioni principali

- Le due principali operazioni da realizzare sulle liste concatenate (sui *nodi* della lista nella fase di implementazione) sono: la *cancellazione* di un elemento della lista e l'*inserimento* di un elemento in una qualsiasi posizione della lista. Quindi diminuire o aumentare la lunghezza di una lista.

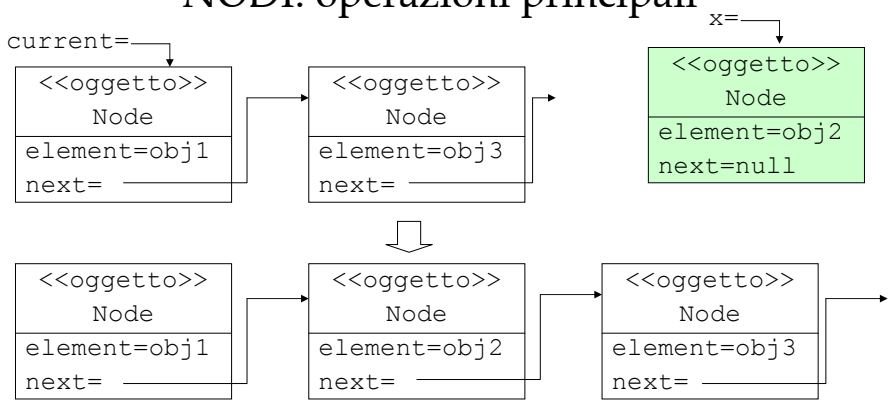
- La semplicità di tali operazioni è la ragione d'essere delle liste concatenate.

NODI: operazioni principali



Il nodo da cancellare viene fatto "*cadere*": `current` è un oggetto `Node`, se prima della cancellazione `current.next` punta al nodo `obj2`, allora: `current.next=current.next.next;`

NODI: operazioni principali



Un nodo da <u>inserire</u>: `current` e `x` sono oggetti `Node`, se prima dell'inserimento `current.next` punta al nodo `obj3`, allora:

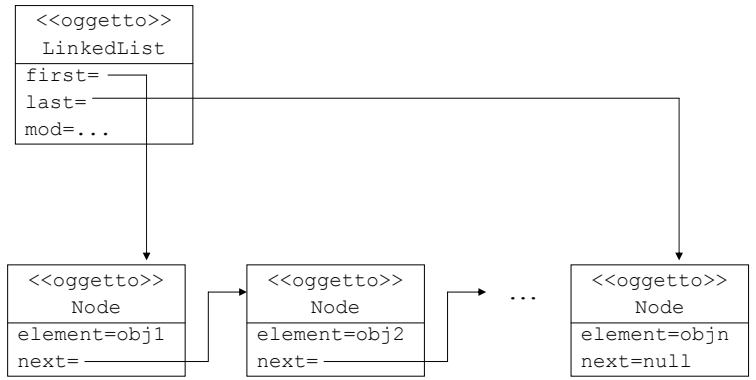
```
x.next=current.next;
current.next=x;
```


LIST: implementazione, LinkedList

- Vediamo una possibile implementazione della lista.
- Si usa una variabile `first` che mantiene il riferimento al primo nodo della lista e una variabile `last` che mantiene un riferimento all'ultimo nodo.
- Inoltre è utilizzata una variabile intera `mod`, che tiene conto delle modifiche alla lista, per evitare che modifiche concorrenti possano interferire con il comportamento degli iteratori.

LINKEDLIST

- Una possibile rappresentazione grafica:



LINKEDLIST

```
public class LinkedList implements List {
  Node first;
  Node last;
  int mod;

  public LinkedList(){
     first=last=null;
     mod=0;
  }
```

Numero di modifiche: chiamate a `remove()`, `add()` e `clear()`.

LINKEDLIST

```
public boolean isEmpty() {
     return first==null;
  }

  public void clear() {
     first=last=null;
     mod++;
  }
```


LINKEDLIST: ricerca

```
public boolean contains(Object x) {
  if (x==null)
     throw new IllegalArgumentException();
  for (Node i=first; i!=null; i=i.next)
     if (x.equals(i.element))
          return true;
  return false;
  }
```

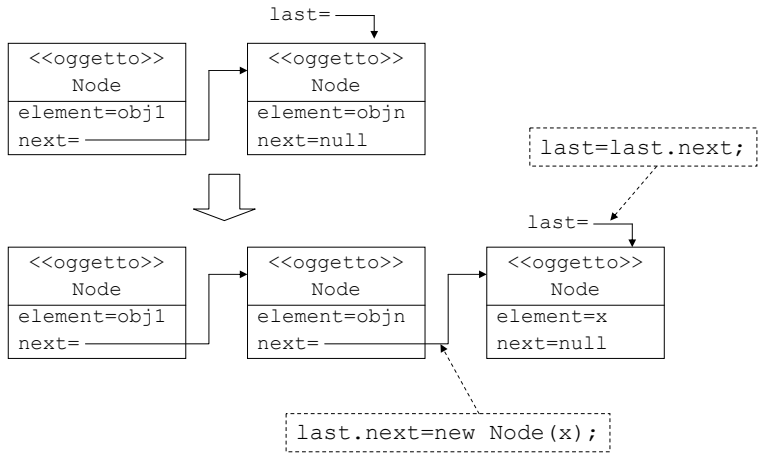
Si esaminano i nodi della lista con un ciclo `for`: si definisce una variabile `i` di tipo `Node` a cui si assegna un riferimento al primo nodo. Si esce dal ciclo quando si finiscono i nodi. L'incremento si ottiene con l'assegnamento `i=i.next`, che sposta il riferimento al nodo successivo.

LINKEDLIST: inserimento

```
public boolean add(Object x) {
  if (x==null)
     throw new IllegalArgumentException();
  if (isEmpty()){      ← Lista vuota
     first=last=new Node(x);
  }else{               ← Lista non vuota
     last.next=new Node(x);
     last=last.next;
  }
  mod++;
  return true;
  }
```


LINKEDLIST: inserimento

- Lista non vuota:



LINKEDLIST: prestazioni

- Aggiungere un nodo alla lista ha un costo computazionale costante *O(1)*: è indipendente dal numero di nodi nella lista. Non si devono copiare gli elementi in una nuova struttura, come nel caso di array pieni.
- Verificare se la lista contiene un dato oggetto implica l'uso di un ciclo `for`. Si consideri il caso medio: se ogni nodo della lista ha la stessa probabilità di contenere il dato, allora si ha una iterazione per verificare il primo nodo, due iterazioni per verificare il secondo nodo, e in una sequenza lunga `n` si ha in media

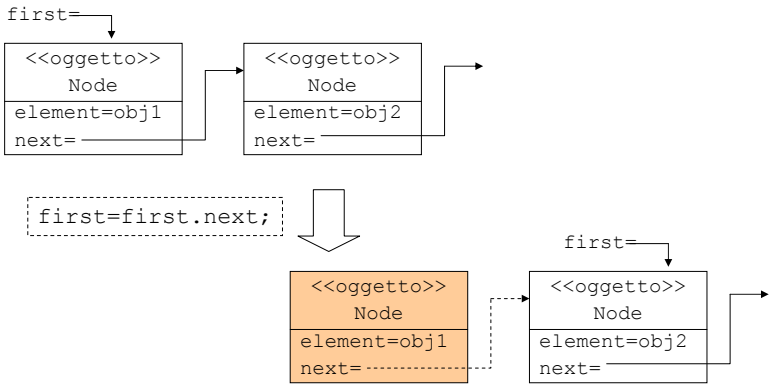
$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = O(n)$$

LINKEDLIST: rimozione

```
public boolean remove(Object x) {
       if (x==null)     throw new IllegalArgumentException();
       if (isEmpty())   return false;
       if (x.equals(first.element)){ <-- Primo elemento
              if (first==last)       <-- e ultimo
                     last=null;
              first=first.next;
              mod++;
              return true;
       }
       for(Node i=first; i.next!=null; i = i.next){
              if(x.equals(i.next.element)){
                     if(i.next==last)
                            last=i;
                     i.next=i.next.next;
                     mod++;
                     return true;
              }
       }
       return false;}
```


LINKEDLIST: rimozione

- Primo elemento:



LINKEDLIST: prestazioni

- Considerazioni analoghe a quelle sviluppate per calcolare la complessità computazionale media del metodo per la ricerca di un dato valgono anche per il metodo di rimozione di un dato.
- Quindi il metodo `remove()` ha in media un costo lineare, cioè *O(n)*.